1. Thomas: Thomas storytelling intro on Alistair and J. Chassaing
    1. Thomas finished by opening a few questions, but without answering them:
        a. What is this alternative?
        b. Can we still talk about hexagonal architecture?
        c. What are the advantages/disadvantages?
        d. When is it useful to use it?
    2. Introducing ourselves: Thomas, Bruno
    3. Bruno: Presentation of the process:
        a. A reminder of what the hexagonal architecture is, the presentation of what the functional core is
        b. We will start from a codebase in hexagonal architecture, and we will transform it together into a functional core / imperative shell
        c. We will finish by comparing this pattern with the hexagonal architecture
    4. Bruno, who presents / Thomas, who comments, questions, clarifies the Theoretical part.
        a. Hexagonal arch (slide)
        b. Functional core / imperative shell (slides)
    5. Thomas, who handles the IDE / Bruno who comments, questions, specifies: Practical part
        a. Presentation of the solution and the hexagon (focus on tests and the web controller)
        b. We take the edge effects out of our hexagon
        c. We see that the method of our web controller is a shell imperative, but
        d. Bruno's remark:
            1. It's easy because our business code was already immutable (and yes, we do DDD and use value types whenever possible)
            2. Our imperative shell is not super functional oriented. We can do better with the Maybe () functor
    6. Bruno, who gets the hand and the code / Thomas who comments, questions, specifies:
        a. Introduction of Maybe: what it is, why it is useful to use it
        b. Copy and paste the Maybe from Mark Seeman's site
        c. Setting up. Explanation of what happens in terms of call kinematics (for those who do not do C #)
        d. Discussion of its interest when there are even more calls to functions of our core (we change the model: instead of having a single call to our Domain, we chain the calls of different functions that belong to our Domain
        e. reviewed Thomas diagram with call workflow steps

7. Discussion:

1. What does not seem to be an option, for example, in Haskell, can become so in your APIs in java, C #, Kotlin, etc.
2. Thomas: Be careful, an important point: the Domain slightly leaks the infra layer because the sequencing of the function calls is done from outside the functional core
3. My heuristic is, therefore, to only use it when there is not too much I / Os orchestration to do
4. Bruno: it can only work if your Domain is coded in a functional style. Here, our business logic had no state; the state was injected at the start when processing each request
5. Comparative matrix.

8. Conclusion:

1. Try not to stick to just the patterns you know. Dig, see what exists next, but do not choose a model by mode without being able to justify its use in your context. Be able to explain your design choices.